Scheduling tree-shaped task graphs to minimize memory and makespan

Loris Marchal
CNRS and University of Lyon
Lyon, France
loris.marchal@ens-lyon.fr

Oliver Sinnen
University of Auckland
Auckland, New Zealand
o.sinnen@auckland.ac.nz

Frédéric Vivien
INRIA and University of Lyon
Lyon, France
frederic.vivien@inria.fr

***Abstract*—This paper investigates the execution of tree-shaped task graphs using multiple processors. Each edge of such a tree represents a large IO file. A task can only be executed if all input and output files fit into memory, and a file can only be removed from memory after it has been consumed. Such trees arise, for instance, in the multifrontal method of sparse matrix factorization. The maximum amount of memory needed depends on the execution order of the tasks. With one processor the objective of the tree traversal is to minimize the required memory. This problem was well studied and optimal polynomial algorithms were proposed.**

Here, we extend the problem by considering multiple processors, which is of obvious interest in the application area of matrix factorization. With the multiple processors comes the additional objective to minimize the time needed to traverse the tree, i.e., to minimize the makespan. Not surprisingly, this problem proves to be much harder than the sequential one. We study the computational complexity of this problem and provide an inapproximability result even for unit weight trees. Several heuristics are proposed, each with a different optimization focus, and they are analyzed in an extensive experimental evaluation using realistic trees.

***Keywords*-scheduling; makespan-memory tradeoff; tree-shaped task-graphs.**

I. Introduction

Parallel workloads are often modeled as task graphs, where nodes represent tasks and edges represent the dependencies between tasks. There is an abundant literature on task graph scheduling when the objective is to minimize the total completion time, or Makespan. However, as the size of the data to be processed is increasing, the memory footprint of the application must be optimized as it can have a dramatic impact on the algorithm execution time. This is best exemplified with an application which, depending on the way it is scheduled, will either fit in the memory, or will require the use of swap mechanisms or out-of-core techniques. There are very few existing studies on the minimization of the memory footprint when scheduling task graphs, and even fewer of them targeting parallel systems.

We consider the following memory-aware parallel scheduling problem for rooted trees. The nodes of the tree correspond to tasks, and the edges correspond to the dependencies among the tasks. The dependencies are in the form of input and output files: each node takes as input several large files, one for each of its children, and it produces a single large file, and the different files may have different sizes. Furthermore, the execution of any node requires its *execution* file to be present; the execution file can be seen as the program of the task. We are to execute such a set of tasks on a parallel system made of p identical processing resources sharing the same memory. The execution scheme corresponds to a schedule of the tree where processing a node of the tree translates into reading the associated input files and producing the output file. How can the tree be scheduled so as to optimize the memory usage?

Modern computing platforms exhibit a complex memory hierarchy ranging from caches to RAM and disks and even sometimes tape storage, with the classical property that the smaller the memory, the quicker. Thus, to avoid large running times, one usually wants to avoid the use of memory devices whose IO bandwidth is below a given threshold: even if out-of-core execution (when large data are unloaded to disks) is possible, this requires special care when programming the application and one usually wants to stay in the main memory (RAM). This is why in this paper, we are interested in the question of minimizing the amount of *main memory* needed to completely process an application.

Throughout the paper, we consider *in-trees* where a task can be executed only if all its children have already been executed. (This is absolutely equivalent to considering *out-trees* as a solution for an in-tree can be transformed into a solution for the corresponding out-tree by just reversing the arrow of time, as outlined in [1].) A task can be processed only if all its files (input, output, and execution) fit in currently available memory. At a given time, many files may be stored in the memory, and at most p tasks may be processed by the p processors. This is obviously possible only if all tasks and execution files fit in memory. When a task finishes, the memory needed for its execution file and its input files is released. Clearly, the schedule which determines the processing times of each task plays a key role in determining which amount of main memory is needed for a successful execution of the whole tree.

The first motivation for this work comes from numerical linear algebra. Tree workflows (assembly or elimination trees) arise during the factorization of sparse matrices, and the huge size of the files involved makes it absolutely necessary to reduce the memory requirement of the factorization. The sequential version of this problem (i.e., with $p = 1$ processor) has already been studied. Liu [2] discusses how to find a memory-minimizing traversal when the traversal is required to correspond to a postorder traversal of the tree. In the follow-up study [3], an exact algorithm is shown to solve the problem, without the postorder constraint on the traversal. Recently, some of us [1] proposed another algorithm to find a memory-optimal traversal, which proved to be faster on existing elimination trees, although being of the same worst-case complexity ($O(n^2)$).

The parallel version of this problem is a natural continuation of these studies: when processing large elimination trees, it is very meaningful to take advantage of parallel processing resources. However, to the best of our knowledge, there exist no theoretical studies for this problem. The key contributions of this work are:

- The proof that the parallel variant of the *pebble game* problem is NP-complete. This shows that the introduction of memory constraints, in the simplest cases, suffices to make the problem NP-hard.
- The proof that no algorithm can simultaneously deliver a constant-ratio approximation for the memory minimization and for the makespan minimization.
- A set of heuristics having different optimizing focus.
- An exhaustive set of simulations using realistic tree shaped task graphs to assess the relative and absolute performance of these heuristics.

The rest of this paper is organized as follows. Section II reviews related studies. The notation and formalization of the problem are introduced in Section III. Complexity results are presented in Section IV while Section V proposes different heuristics to solve the problem, which are evaluated in Section VI.

II. Background and Related Work

A. Sparse matrix factorization

As mentioned above, determining a memory-efficient tree traversal is very important in sparse numerical linear algebra. The elimination tree is a graph theoretical model that represents the storage requirements, and computational dependencies and requirements, in the Cholesky and LU factorization of sparse matrices. In a previous study, we have described how such trees are built, and how the multifrontal method organizes the computations along the tree [1]. This is the context of the founding studies of Liu [2], [3] on memory minimization for postorder or general tree traversals presented in the previous section. Memory minimization is still a concern in modern multifrontal solvers when dealing with large matrices. Among other, efforts have been made to design dynamic schedulers that takes into account dynamic pivoting (which impacts the weights of edges and nodes) when scheduling elimination trees with strong memory constraints [4], or to consider both task and tree parallelism with memory constraints [5]. While these studies try to optimize memory management in existing parallel solvers, we aim at designing a simple model to study the fundamental underlying scheduling problem.

B. Scientific workflows

The problem of scheduling a task graph under memory constraints also appears in the processing of scientific workflows whose tasks require large I/O files. Such workflows arise in many scientific fields, such as image processing, genomics or geophysical simulations. The problem of task graphs handling large data has been identified in [6] which proposes some simple heuristic solutions. Surprisingly, in the context of quantum chemistry computations, Lam et al. [7] have recently rediscovered the algorithm published in 1987 in [3].

C. Pebble game and its variants

On the more theoretical side, this work builds upon the many papers that have addressed the pebble game and its variants. Scheduling a graph on one processor with the minimal amount of memory amounts to revisiting the I/O pebble game with pebbles of arbitrary sizes that must be loaded into main memory before *firing* (executing) the task. The pioneering work of Sethi and Ullman [8] deals with a variant of the pebble game that translates into the simplest instance of our problem when all input/output files have weight 1 and all execution files have weight 0. The concern in [8] was to minimize the number of registers that are needed to compute an arithmetic expression. The problem of determining whether a general DAG can be executed with a given number of pebbles has been shown NP-hard by Sethi [9] if no vertex is pebbled more than once (the general problem allowing recomputation, that is, re-pebbling a vertex which have been pebbled before, has been proven PSPACE complete [10]). However, this problem has a polynomial complexity for tree-shaped graphs [8].

To the best of our knowledge, there have been no attempts to extend these results to parallel machines, with the objective of minimizing both memory and total execution time. We present such an extension in Section IV.

III. Model and objectives

A. Application model

We consider in this paper a tree-shaped task-graph T composed of n nodes, or tasks, numbered from 1 to n. Nodes in the tree have an output file, an execution file (or program), and several input files (one per child). More precisely:

- Each node i in the tree has an execution file of size n_i and its processing on a processor takes time w_i.

- Each node i has an output file of size f_i. If i is not the root, its output file is used as input by its parent $parent(i)$; if i is the root, its output file can be of size zero, or contain outputs to the outside world.
- Each non-leaf node i in the tree has one input file per child. We denote by $Children(i)$ the set of the children of i. For each child $j \in Children(i)$, task j produces a file of size f_j for i. If i is a leaf-node, then $Children(i) = \emptyset$ and i has no input file: we consider that the initial data of the task either reside in its execution file or are read from disk (or received from the outside word) during the execution of the task.

During the processing of a task i, the memory must contain its input files, the execution file, and the output file. The memory needed for this processing is thus:

$$\left(\sum_{j \in Children(i)} f_j \right) + n_i + f_i$$

After i has been processed, its input files and program are discarded, while its output file is kept in memory until the processing of its parent.

B. Platform model and objectives

In this paper, our goal is to design the simpler platform model which allows to study memory minimization on a parallel platform. We thus consider p identical processors which share a single memory. We do not consider here a hard constraint on the memory, but we rather include memory in the objectives. We thus consider multi-criteria optimization with the following two objectives:

- **Makespan.** Our first objective is the classical makespan, or total execution time, which corresponds to the times-span between the beginning of the execution of the first leaf task and the end of the processing of the root task.
- **Memory.** Our second objective is the amount of memory needed for the computation. At each time step, some files are stored in the memory and some task computations occur, which induces a memory usage. The *peak memory* is the maximum usage of the memory over the whole schedule, which we aim at minimizing.

IV. Complexity results in the Pebble Game model

Since there are two objectives, the decision version of our problem can be stated as follows.

Definition 1 (BiObjectiveParallelTreeScheduling)**.** *Given a tree-shaped task graph T provided with memory weights and task durations, p processors, and two bounds $B_{C_{\max}}$ and B_{mem}, is there a schedule of the task graph on the processors whose makespan is not larger than $B_{C_{\max}}$ and whose peak memory is not larger than B_{mem}?*

This problem is obviously NP-complete. Indeed, when there are no memory constraints ($B_{mem} = \infty$) and when the task tree does not contain any inner node, that is, when all tasks are either leaves or the root, then our problem is equivalent to scheduling independent tasks on a parallel platform which is an NP-complete problem as soon as tasks have different execution times [11]. On the contrary minimizing the makespan for a tree of same-size tasks can be solve in polynomial-time when there are no memory constraints [12]. In this section, we consider the simplest variant of the problem. We assume that all input files have the same size ($\forall i, f_i = 1$) and no extra memory is needed for computation ($\forall i, n_i = 0$). Furthermore, we assume that the processing of each node takes a unit time: $\forall i, w_i = 1$. We call this variant of the problem the *Pebble Game* model since it perfectly corresponds to pebble game problems introduced above: the weight $f_i = 1$ corresponds to the pebble put on one node once it has been processed and its results is not yet discarded. Processing a node requires to put an extra pebble on this node and is done in unit time.

In this section, we first show that even in this simple variant, the introduction of memory constraints (a limited number of pebbles) makes the problem NP-hard (Section IV-A). Then, we show that when trying to minimize both memory and makespan, it is not possible to get a solution with a constant approximation ratio for both objectives (Section IV-B).

A. NP-completeness

Theorem 1. *The BiObjectiveParallelTreeScheduling problem is NP-complete in the Pebble Game model (i.e., with $\forall i, f_i = w_i = 1, n_i = 0$).*

Proof: First, it is straightforward to check that the problem is in NP: given a schedule, it is easy to compute its peak memory and makespan.

To prove the problem NP-completeness, we perform a reduction from 3-Partition, which is known to be NP-complete in the strong sense [13]. We consider the following instance $\mathcal{I}_1$ of the 3-Partition problem: let a_i be $3m$ integers and B an integer such that $\sum a_i = mB$. We consider the variant of the problem, also NP-complete, where $\forall i, B/4 < a_i < B/2$. To solve $\mathcal{I}_1$, we need to solve the following question: does there exist a partition of the a_i's in m subsets $S_1, \ldots, S_m$, each containing exactly 3 elements, such that, for each S_k, $\sum_{i \in S_k} a_i = B$. We build the following instance $\mathcal{I}_2$ of our problem, illustrated on Figure 1. The tree contains a root r with $3m$ children, the N_i's, each one corresponding to a value a_i. Each node N_i has $3m \times a_i$ children, which are leaf nodes. The question is to find a schedule of this tree on $p = 3mB$ processors, whose peak memory is not larger than $B_{mem} = 3m \times B + 3m$ and whose makespan is not larger than $B_{C_{\max}} = 2m + 1$.

Assume first that there exists a solution to $\mathcal{I}_1$, i.e., that there are m subsets S_k of 3 elements with $\sum_{i \in S_k} a_i = B$. In this case, we build the following schedule:

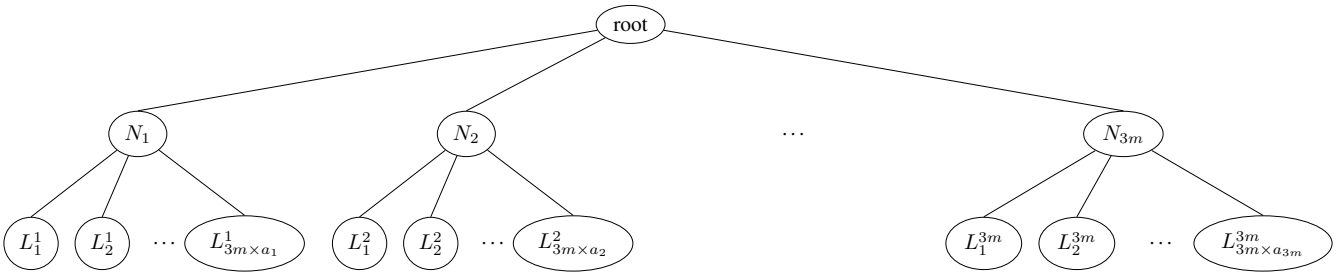

Figure 1. Tree used for the NP-completeness proof

- At step 1, we process all the nodes $L_x^{i_1}$, $L_y^{j_1}$, and $L_z^{k_1}$ with $S_1 = \{a_{i_1}, a_{j_1}, a_{k_1}\}$. There are $3mB = p$ such nodes, and the amount of memory needed is also $3mB$.
- At step 2, we process the nodes N_{i_1}, N_{j_1}, N_{k_1}. The memory needed is $3mB + 3$.
- At step $2n + 1$, with $1 \leq n \leq m - 1$, we process the $3mB = p$ nodes $L_x^{i_n}$, $L_y^{j_n}$, $L_z^{k_n}$ with $S_n = \{a_{i_n}, a_{j_n}, a_{k_n}\}$. The amount of memory needed is $3mB + 3n$ (counting the memory for the output files of the N_t nodes previously processed).
- At step $2n + 2$, with $1 \leq n \leq m - 1$, we process the nodes N_{i_n}, N_{j_n}, N_{k_n}. The memory needed for this step is $3mB + 3(n + 1)$.
- At step $2m + 1$, we process the root node and the memory needed is $3m + 1$.

Thus, the peak memory of this schedule is B_{mem} and its makespan $B_{C_{\max}}$.

On the contrary, assume that there exists a solution to problem $\mathcal{I}_2$, that is, that there exists a schedule of makespan at most $B_{C_{\max}} = 2m + 1$. Without loss of generality, we assume that the makespan is exactly $2m + 1$. We start by proving that at any step of the algorithm there are at most three of the N_i nodes that are processed. By contradiction, assume that four (or more) such nodes $N_{i_1}, N_{i_2}, N_{i_3}, N_{i_4}$ are processed during a certain step. We recall that $a_i > B/4$ so that $a_{i_1} + a_{i_2} + a_{i_3} + a_{i_4} > B$ and thus $a_{i_1} + a_{i_2} + a_{i_3} + a_{i_4} \geq B + 1$. The memory needed at this step is thus at least $(B + 1)3m$ for the children of the nodes N_{i_1}, N_{i_2}, N_{i_3}, and N_{i_4} and 4 for the nodes themselves, hence a total of at least $(B+1)3m+4$, which is more than the prescribed bound B_{mem}. Thus, at most three of N_i nodes are processed at any step. In the considered schedule, the root node is processed at step $2m+1$. Then, at step $2m$, some of the N_i nodes are processed, and at most three of them from what precedes. The a_i's corresponding to those nodes make the first subset S_1. Then all the nodes L_x^j such that $a_j \in S_1$ must have been processed at the latest at step $2m-1$, and they occupy a memory footprint of $3m \sum_{a_j \in S_1} a_j$ at steps $2m - 1$ and steps $2m$. Let us assume that a node N_k is processed at step $2m-1$. For the memory bound B_{mem} not to be satisfied we must have $a_k + \sum_{a_j \in S_1} a_j \leq B$. (Otherwise, we would need a memory of at least $3m(B + 1)$ for the involved L_x^j nodes plus 1 for the node N_k). Therefore, node N_k could have been processed at step $2m$. We then modify the schedule so as to schedule N_k at step $2m$ and thus we add k to S_1. We can therefore assume, without loss of generality, that no N_i node is processed at step $2m-1$. Then, at step $2m-1$ only children of the N_j nodes with $a_j \in S_1$ are processed, and all of them are. So, none of them have any memory footprint before step $2m-1$. We then generalize this analysis: at step $2i$, for $1 \leq i \leq m - 1$, only some N_j nodes are processed and they define a subset S_i; at step $2i-1$, for $1 \leq i \leq m-1$, are processed exactly the nodes L_x^k that are children of the nodes N_j such that $a_j \in S_i$.

Because of the memory constraint, each of the m subsets of a_i's built above sum to at most B. Since they contain all a_i's, their sum is mB. Thus, each subset S_k sums to B and we have built a solution for $\mathcal{I}_1$. ■

B. Joint minimization of both objectives

As our problem is NP-complete, it is natural to wonder whether there exist approximation algorithms. Here, we prove that there does not exist schedules which approximates both the minimum makespan and the minimum memory with constant factors[1].

Theorem 2. *There is no algorithm that is both an α-approximation for makespan minimization and a β-approximation for memory peak minimization when scheduling in-tree task graphs.*

Below is a sketch of the proof of Theorem 2. A comprehensive proof can be found in the companion research report [14].

Proof: To establish this result, we proceed by contradiction. We therefore assume that there is an integer α, an integer β, and an algorithm $\mathcal{A}$ that processes any input tree $\mathcal{T}$ in a time not greater than α times the optimal execution time while using a peak memory that is not greater than β times the optimal peak memory.

The tree. Figure 2 presents the tree used to derive a contradiction. This tree is made of n identical subtrees

[1]This is equivalent to say that there is no *Zenith* or *simultaneous* approximation.

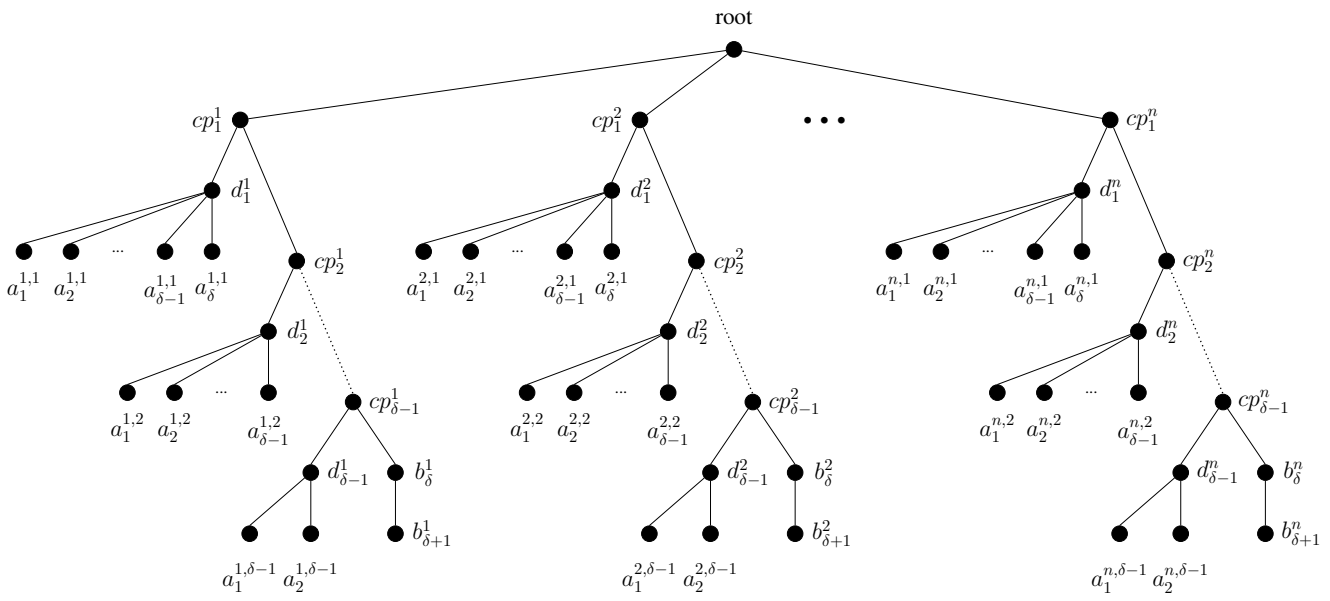

Figure 2. Tree used for establishing Theorem 2.

whose roots are the children of the tree root. The values of n and δ will be fixed later on.

Optimal execution time. The optimal execution time is equal to the length of the critical path, as we have made no hypothesis on the number of available processors. The critical path has a length of $\delta+2$, which is the length of the path from the root to any $b_{\delta+1}^i$, $a_1^{i,\delta-1}$, or $a_2^{i,\delta-1}$ node, with $1 \leq i \leq n$.

Optimal peak memory. Let us consider any sequential execution that is optimal with regard to the peak memory usage. Under this execution, let d_1^i be the last processed node among the d_1^j nodes, $1 \leq j \leq n$. We consider the step at which node d_1^i is processed. As, by hypothesis, all the d_1^j nodes, $1 \leq j \leq n$ and $j \neq i$, have already been processed, there are in memory at that step at least $n-1$ results. The processing of d_1^i requires $\delta+1$ memory units as this node has δ children. Hence, a total memory usage of at least $(n-1)+(\delta+1) = \delta+n$ for the processing of d_1^i. This is obviously a lower bound on the optimal peak memory usage. We now show that this bound can be reached.

We consider the following schedule:

- Completely process first the subtree rooted at cp_1^1, then the subtree rooted at cp_1^2, and so on.
- The subtree rooted at cp_1^i is processed as follows: for j going from 1 to $\delta-1$, process the $\delta-j+1$ children of node d_i^j, then node d_i^j; then process nodes $b_{\delta+1}^i$, b_δ^i, and nodes $cp_{\delta-1}^i$ to cp_1^i.

Under this schedule, the peak memory usage during the processing of the subtree rooted at cp_1^i is $i+\delta$. The overall peak memory usage of the studied schedule is then $n+\delta$ which is thus the optimal peak memory usage.

Lower bound on the peak memory usage of $\mathcal{A}$. The peak memory usage is not smaller than the average memory usage. We derive the desired contradiction by using the average memory usage of algorithm $\mathcal{A}$ as a lower bound to its peak memory usage.

By hypothesis, algorithm $\mathcal{A}$ is α competitive with regard to makespan minimization. Therefore the processing of the tree by algorithm $\mathcal{A}$ should complete at the latest at time $\alpha(\delta+2)$. To ensure that, the n cp_1^i nodes, $1 \leq i \leq n$, must all be executed at the latest at time $\alpha(\delta+2)-1$. Therefore, all the descendants of these nodes must be executed between time 0 and time $\alpha(\delta+2)-2$. All together, the nodes cp_1^i, for $1 \leq i \leq n$ have $n\frac{\delta^2+5\delta-4}{2}$ descendants.

We consider the memory footprint of each of these nodes between time step 0 and time step $\alpha(\delta+2)-2$. The result of the processing of each of theses nodes must be in memory for at least two steps in this interval, the step at which the node is processed and the step at which its parent node is processed, except for the nodes d_j^1, $1 \leq j \leq n$, and cp_2^k, for $1 \leq k \leq n$, whose parents need not have been processed in that interval and thus need only to be present in memory during one time step. The overall memory footprint between time 0 and $\alpha(\delta+2)-2$ is then: $n\left(\delta^2+5\delta-6\right)$. The average memory usage during that period is thus:

$$\frac{n\left(\delta^2+5\delta-6\right)}{\alpha(\delta+2)-2}.$$

This is obviously a lower bound on the overall peak memory usage. This bound enables us to derive a lower bound lb on the approximation ratio ρ of algorithm $\mathcal{A}$ with regard to

memory usage:

$$\rho \geq lb = \frac{\frac{n\left(\delta^2+5\delta-6\right)}{\alpha(\delta+2)-2}}{n+\delta} = \frac{n\left(\delta^2+5\delta-6\right)}{(\alpha(\delta+2)-2)(n+\delta)}.$$

We then let $\delta = n^2$. Therefore,

$$lb = \frac{n\left(n^4+5n^2-6\right)}{(\alpha(n^2+2)-2)(n+n^2)}.$$

Then, lb tends to $+\infty$ when n tends to infinity. There is thus a value n_0 such that, for any value $n \geq n_0$, the right-hand side is greater than 2β. We let $n = n_0$ and we obtain:

$$lb = \frac{n_0\left(n_0^4+5n_0^2-6\right)}{(\alpha(n_0^2+2)-2)(n_0+n_0^2)} \geq 2\beta,$$

which contradicts the definition of β. ∎

V. Heuristics

Given the complexity of optimizing the makespan and memory at the same time, we have investigated heuristics and propose three algorithms: ParSubtrees, ParInnerFirst, and ParDeepestFirst. The intention is that the proposed algorithms cover a range of use cases, where the optimization focus wanders between the makespan and the required memory. ParSubtrees employs a memory-optimizing sequential algorithm for its subtrees, hence its focus is more on the memory side. In contrast, ParInnerFirst and ParSubtrees are list scheduling based algorithms, which should be stronger in the makespan objective. Nevertheless, ParInnerFirst tries to approximate a postorder in parallel, which is good for memory in sequential. ParDeepestFirst's focus is fully on the makespan.

The minimal memory requirement M is achieved by using the optimal sequential algorithm [1], i.e., using $p = 1$ processor. Employing more processors cannot reduce the amount of memory required, yet the sequential algorithm is of course only a p-approximation of the optimal parallel makespan C^*_{max}.

A. Heuristic ParSubtrees

The most natural idea to process a tree T in parallel is arguably its splitting into subtrees and their subsequent parallel processing, each using the sequentially memory-optimal algorithms [1], [3]. An underlying idea is to give each processor a whole subtree in order to enable a lot of parallelism while allowing to use single-processor memory-optimal traversals on each subtree. Algorithm 1 outlines such an algorithm, together with the routine for splitting T into subtrees given in Algorithm 2. The makespan obtained using ParSubtrees is denoted by $C^{\text{ParSubtrees}}_{max}$.

In this approach, q subtrees of T, $q \leq p$, are processed in parallel. Each of these subtrees is a maximal subtree of T. In other words, each of these subtrees includes all the descendants (in T) of its root. The nodes not belonging to the q subtrees are processed sequentially. These are the nodes where the q subtrees merge, the nodes included in subtrees that were produced in excess (if more than p subtrees were created), and the ancestors of these nodes. An alternative approach, as discussed below, is to process all subtrees in parallel, assigning more than one subtree to each processor, but Algorithm 1 allows us to find a *makespan*-optimal splitting into subtrees, established shortly in Lemma 1.

Algorithm 1: ParSubtrees (T, p)

1. Split tree T into q subtrees ($q \leq p$) and remaining set of nodes, using SplitSubtrees (T, p).
2. Concurrently process the q subtrees, each using memory minimizing algorithm, e.g. [1].
3. Sequentially process remaining set of nodes, using memory minimizing algorithm.

As w_i is the computation weight of node i, W_i denotes the total computation weight (i.e., sum of weights) of all nodes in the subtree rooted in i, including i. SplitSubtrees uses a node priority queue PQ in which the nodes are sorted by non-increasing W_i, and ties are broken according to non-increasing w_i. $head(PQ)$ returns the first node of PQ, while $popHead(PQ)$ also removes it. $PQ[i]$ denotes the i-th element in the queue.

SplitSubtrees starts with the root and continues splitting the largest subtree (in terms of W) until this subtree is a leaf node ($W_{head(PQ)} = w_{head(PQ)}$). The execution time of Step 2 of ParSubtrees is that of the largest of the q subtrees, hence $W_{head(PQ)}$ of the splitting. Splitting subtrees that are smaller than the largest leaf ($W_j < \max_{i \in T} w_i$) cannot decrease the parallel time, but only increase the sequential time. More generally, given any splitting s of T into subtrees, the best execution time for s with ParSubtrees is achieved by choosing the p largest subtrees for the parallel Step 2. This can be easily derived, as swapping a large tree included in the sequential part with a smaller tree included in the parallel part cannot increase the total execution time.

Lemma 1. SplitSubtrees *returns a splitting of T into subtrees that results in the makespan-optimal processing of T with* ParSubtrees.

Proof: The proof is by contradiction. Let S be the splitting into subtrees selected by SplitSubtrees. Assume now that there is a different splitting S_{opt} which results in a shorter processing with ParSubtrees.

Let r be the root node of a heaviest subtree in S_{opt}. Let t be the first step in SplitSubtrees where a node, say r_t, of weight W_r is the head of PQ at the end of the step (r_t is not necessarily equal to r, as there can be more than one subtree of weight W_r). There is always such a step t, because all subtrees are split by SplitSubtrees until at least one of the largest trees is a leaf node. By definition of r, there cannot be any leaf node heavier than W_r. The cost

Algorithm 2: SPLITSUBTREES (T, p)

Compute weights $W_i, \forall i \in T$
$PQ \leftarrow root$
$seqSet \leftarrow \emptyset$
$Cost(0) = W_{root}$
$s \leftarrow 1$ /* splitting rank */
while $W_{head(PQ)} > w_{head(PQ)}$ **do**
  $node \leftarrow popHead(PQ)$
  $seqSet \leftarrow seqSet \cup node$
  $PQ \leftarrow Children(node)$
  $C_{max}^{\text{ParSubtrees}}(s) = W_{head(PQ)} + \sum_{i \in seqSet} w_i + \sum_{i=PQ[p+1]}^{|PQ|} W_i$
  $s \leftarrow s + 1$
Select splitting x with $C_{max}^{\text{ParSubtrees}}(x) = \min_{t=0}^{s-1} C_{max}^{\text{ParSubtrees}}(t)$

of the solution of step t is $C_{max}^{\text{ParSubtrees}}(t) = W_r + Seq(t)$, hence parallel time plus sequential time, denoted by $Seq(t)$. $Seq(t)$ is the total weight of the sequential set $seqSet$ plus the total weight of the surplus subtrees (that is, of all the subtrees except the p ones of largest weights). The cost of S_{opt} is $C_{max}^{*} = W_r + Seq(S_{opt})$, given that r is the root of a heaviest subtree of S_{opt} by definition.

The splitting at step t (and any other splitting considered by SPLITSUBTREES) cannot be identical to S_{opt}, otherwise SPLITSUBTREES would have selected that splitting. All subtrees that were split in SPLITSUBTREES before step t were strictly heavier than W_r. Thus, there cannot exist any subtree in S_{opt}, whose subtrees are part of the splitting at step t. Hence for every subtree T_j in the splitting at step t the following property holds: either T_j is part of S_{opt} or a splitting of T_j into subtrees is part of S_{opt}. It directly follows that $Seq(t) \leq Seq(S_{opt})$, because every splitting of a tree into subtrees increases the sequential time by at least the root's weight. As the parallel time is identical for t and S_{opt}, namely W_r, it follows that $C_{max}^{\text{ParSubtrees}}(t) \leq C_{max}^{*}$, which is a contradiction to S_{opt}'s shorter processing time. ■

Complexity: We first analyse the complexity of SPLITSUBTREES. Computing the weights W_i costs $O(n)$. Each insertion into PQ costs $O(\log(n))$ and calculating $C_{max}^{\text{ParSubtrees}}(s)$ in each step costs $O(p)$. Given that there are $O(n)$ steps, SPLITSUBTREES's complexity is $O(n(\log(n)+p))$. The complexity of the sequential traversal algorithms used in Steps 2 and 3 of PARSUBTREES cost at most $O(n^2)$, e.g., [1], [3], or $O(n\log(n))$ if the optimal postorder suffices. Thus the total complexity of PARSUBTREES is $O(n^2)$ or $O(n\log(n))$, depending on the chosen sequential algorithm.

PARSUBTREES has the following guarantees for the memory requirement and makespan.

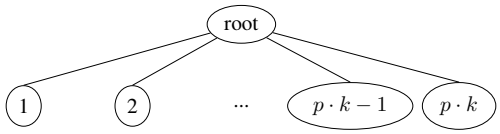

Figure 3. PARSUBTREES is at best a p-approximation for the makespan.

Memory: PARSUBTREES is a $(p+1)$-approximation algorithm for peak memory minimization. During the parallel part of PARSUBTREES the total memory used is less than p times the memory for the complete sequential execution (M_{seq}), $M_p \leq p \cdot M_{seq}$. This is because each of the p processors executes a maximal subtree and that the processing of any subtree uses, obviously, less memory (if done optimally) than the processing of the whole tree. During the sequential part of PARSUBTREES the memory is bounded by $M_s \leq M_{seq} + p \cdot \max_{i \in T} f_i \leq (p+1)M_{seq}$, where the second term is for the output files produced by the up to p subtrees processed in parallel. Hence, in total: $M \leq (p+1)M_{seq}$

Makespan: PARSUBTREES delivers a p-approximation algorithm for makespan minimization. In other words, the makespan achieved by PARSUBTREES can be up to p times worse than the optimal makespan and thus may be not faster than the sequential execution. This can be derived readily with a tree of height 1 and $p \cdot k$ leaves (a fork) and $w_i = 1, \forall i \in T$, where k is a large integer (this tree is depicted on Figure 3). The optimal makespan for such a tree is $C_{max}^{*} = kp/p + 1 = k + 1$. With PARSUBTREES the makespan is $C_{max} = 1 + (1 + pk - p) = p(k-1) + 2$. When k tends to $+\infty$ the ratio between the makespans tends to p.

Given the just observed worst case for the makespan, a makespan optimization for PARSUBTREES is to allocate all produced subtrees to the p processors instead of only p. This can be done by ordering the subtrees by non-increasing total weight and allocating each subtree in turn to the processor with the lowest total weight. Each of the parallel processors executes its subtrees sequentially. This optimized form of the algorithm shall be named PARSUBTREESOPTIM. Note that this optimization should improve the makespan, but it will likely worsen the peak memory usage.

B. Heuristic PARINNERFIRST

PARSUBTREES is a high level algorithm employing sequential memory-optimized algorithms. An alternative is to design algorithms that directly work on the tree in parallel and we present two such algorithms. From the sequential case it is known that a *postorder* traversal, while not optimal for all instances, provides good results [1]. Our intention is to extend the principle of postorder traversal to the parallel processing. To do so we establish the following rules.

Parallel Postorder:

1) If an inner node (i.e., a non-leaf node) is ready to be processed (i.e., its input files are all in memory) then execute it.

2) Otherwise, select and process the leaf node that is closest (in terms of edges to be traversed) to the previously selected leaf.

These rules do not correspond to the usual formulation of postorder but, when applied using a single processor, they give rise to a postorder traversal of the tree. Due to the concurrent processing of nodes with p processors, the resulting order will not be a perfect postorder, but hopefully a close approximation.

With the careful formulation of the parallel postorder we are able to base the heuristic on an event-based list scheduling algorithm [15]. Algorithm 3 outlines a generic list scheduling, driven by node finish time events. At each event at least one node has finished so at least one processor is available for processing nodes. Each available processor is given the respective head node of the priority queue.

Algorithm 3: List scheduling(T, p, O)

```
1  Insert leaves in PQ, ordered as in O
2  eventSet ← {0}          /* ascending order */
3  while eventSet ≠ ∅ do          /* event:node
   finishes */
4      popHead(eventSet)
5      Insert new ready nodes in PQ    /* available
       parents of nodes completed at event
       */
6      P_a ← available processors
7      while P_a ≠ ∅ and PQ ≠ ∅ do
8          proc ← popHead(P_a);
           node ← popHead(PQ)
9          Assign node to proc
10         eventSet ← eventSet ∪
           finishTime(node)
```

The order in which nodes are processed in Algorithm 3 is determined by two aspects: i) the node order O given as input; and ii) the ordering established by the priority queue PQ.

For our proposed parallel postorder algorithm, called PARINNERFIRST, the priority queue uses the following ordering: 1) inner nodes, ordered by non-increasing depth; 2) leaf nodes as ordered in the input order O. To achieve a parallel postorder, the node ordering O needs to be a sequential postorder. It makes heuristic sense that this postorder is an optimal sequential postorder, so that memory consumption can be minimized [2].

Complexity: The complexity of PARINNERFIRST is that of determining the input order O and that of the list scheduling. Computing the optimal sequential postorder is $O(n \log n)$ [2]. In the list scheduling algorithm there are $O(n)$ events and n nodes are inserted and retrieved from PQ. An insertion into PQ is $O(\log n)$, so the list scheduling complexity is $O(n \log n)$. Hence, the total complexity is also $O(n \log n)$.

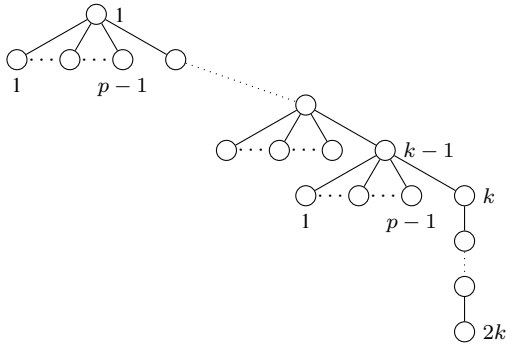

Figure 4. No memory bound for PARINNERFIRST.

In the following we study the memory requirement and makespan of PARINNERFIRST.

Memory: There is no limit on the required memory compared to the optimal sequential memory M_{seq}. This is derived considering the tree in Figure 4. All output files have size 1 and the execution files have size 0 ($f_i = 1, n_i = 0$ for any node i of T). When optimally processing with $p = 1$, we process the leaves in a deepest first order. The resulting optimal memory requirement is $M_{seq} = p+1$, reached when processing a join node.With p processors all leaves have been processed at the time the first join node $(k-1)$ can be executed. (The longest chain has length $2k$.) At that time there are $(k-1) \cdot (p-1) + 1$ files in memory. When k tends to $+\infty$ the ratio between the memory requirements also tends to $+\infty$.

Makespan: PARINNERFIRST schedule is a $(2 - \frac{1}{p})$-approximation algorithm for makespan minimization because PARINNERFIRST is a list scheduling algorithm [16].

C. Heuristic PARDEEPESTFIRST

The previous heuristic PARINNERFIRST is motivated by good memory results for sequential postorder. Going the opposite direction, a heuristic objective can be the minimization of the makespan. For trees, all inner nodes depend on the leaf nodes, so it makes heuristic sense to try to process the deepest nodes first to reduce any possible waiting time. For the parallel processing of the tree, the most meaningful definition of the depth of a node i is the w-weighted length of the path from i to the root of the tree. This path length includes the w_i. The deepest node is the first node of the critical path of the tree.

PARDEEPESTFIRST is our proposed algorithm that does this. Due to the general nature of the list scheduling presented in Algorithm 3, we can implement PARDEEPESTFIRST with it. To achieve the deepest first processing the priority queue PQ orders the nodes as follows: 1) deepest nodes first (in terms of w-weighted path length to root); 2) inner nodes before leaf nodes; 3) leaf nodes are ordered in

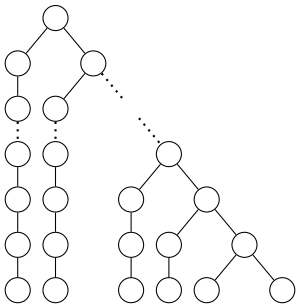

Figure 5. Tree with long chains.

the input order O. Note that the leaf order is only relevant for leaves of the same depth. This order should nevertheless be "reasonable", i.e., it should not alternate between leaves from different parents, which would be bad for the memory consumption. Such an order is again easily achieved when O is a sequential postorder.

Complexity: The complexity is the same as for PARINNERFIRST, namely $O(n \log n)$. See PARINNERFIRST's complexity analysis.

Now we study the memory requirement and the makespan of PARDEEPESTFIRST.

Memory: The required memory of PARDEEPESTFIRST is unbounded compared to the optimal sequential memory M_{seq}. Consider the tree in Figure 5 with many long chains, assuming the Pebble Game model (i.e., $f_i = 1$, $n_i = 0$, and $w_i = 1$ for any node i of T). The optimal sequential memory requirement is 3.The memory usage of PARDEEPESTFIRST will be proportional to the number of leaves, because they are all at the same depth, the deepest one. As we can build a tree like the one of Figure 5 for any predefined number of chains, the ratio between the memory required by PARDEEPESTFIRST and the optimal one is unbounded.

Makespan: PARDEEPESTFIRST schedule is a $(2 - \frac{1}{p})$-approximation algorithm for makespan minimization because PARDEEPESTFIRST is, like PARINNERFIRST, a list scheduling algorithm [16].

VI. EXPERIMENTAL VALIDATION

In this section, we experimentally compare the heuristics proposed in the previous section, and we compare their performance to lower bounds.

A. Setup

All heuristics have been implemented in C. Special care has been devoted to the implementation to avoid complexity issues. Especially, priority queues have been implemented using binary heap to allow for $O(\log n)$ insertion and minimum extraction[2].

[2]The code and the data sets are available online at http://graal.ens-lyon.fr/~lmarchal/scheduling-trees/

Instead of implementing an intricate algorithm with $O(n^2)$ complexity such as Liu's algorithm [3] to obtain minimum sequential memory, we have chosen to estimate this minimum memory using the optimal post-order traversal. We have shown in [1] that this traversal was optimal in 95.8% of the tested cases, with an average increase of 1% with respect to the optimal. This justifies this choice. Since the reference sequential task-graph traversal serves as a basis for ordering nodes in a number of our heuristics, a large complexity would be prohibitive for this first step.

B. Data set

The data set contains assembly trees of a set of sparse matrices obtained from the University of Florida Sparse Matrix Collection (http://www.cise.ufl.edu/research/sparse/matrices/). The chosen matrices satisfy the following assertions: not binary, not corresponding to a graph, square, having a symmetric pattern, a number of rows between 20,000 and 2,000,000, a number of nonzeros per row at least equal to 2.5, and a number of nonzeros per row at most equal to 5,000,000; and each chosen matrix has the largest number of nonzeros among the matrices in its group satisfying the previous assertions. At the time of testing there were 76 matrices satisfying these properties. We first order the matrices using MeTiS [17] (through the MeshPart toolbox [18]) and `amd` (available in Matlab), and then build the corresponding elimination trees using the `symbfact` routine of Matlab. We also perform a relaxed node amalgamation on these elimination trees to create assembly trees. We have created a large set of instances by allowing 1, 2, 4, and 16 (if more than 1.6×10^5 nodes) relaxed amalgamations per node. At the end we compute memory weights and processing times to accurately simulate the matrix factorization: we compute the memory weight n_i of a node as $\eta^2 + 2\eta(\mu - 1)$, where η is the number of nodes amalgamated, and μ is the number of nonzeros in the column of the Cholesky factor of the matrix which is associated with the highest node (in the starting elimination tree); the processing cost w_i of a node is defined as $2/3\eta^3 + \eta^2(\mu - 1) + \eta(\mu - 1)^2$ (these terms corresponds to one gaussian elimination, two multiplications of a triangular $\eta \times \eta$ matrix with a $\eta \times (\mu - 1)$ matrix, and one multiplication of a $(\mu - 1) \times \eta$ matrix with a $\eta \times (\mu - 1)$ matrix). The memory weights f_i of edges are computed as $(\mu - 1)^2$.

The resulting 608 trees contains from 2,000 to 1,000,000 nodes. Their depth ranges from 12 to 70,000 and their maximum degree ranges from 2 to 175,000. Each heuristic is tested on each tree using $p = 2$, 4, 8, 16, and 32 processors. Then the memory and makespan of the resulting schedules are evaluated by simulating a parallel execution.

C. Results

The comparison of the heuristics is summarized in Table I. It shows that PARSUBTREES and PARSUBTREESOPTIM are

Heuristic	Best memory	Within 5% of best memory	Avg. deviation from optimal (seq.) memory	Best makespan	Within 5% of best makespan	Avg. deviation from best makespan
ParSubtrees	81.1 %	85.2 %	133.0 %	0.2 %	14.2 %	34.7 %
ParSubtreesOptim	49.9 %	65.6 %	144.8 %	1.1 %	19.1 %	28.5 %
ParInnerFirst	19.1 %	26.2 %	276.5 %	37.2 %	82.4 %	2.6 %
ParDeepestFirst	3.0 %	9.6 %	325.8 %	95.7 %	99.9 %	0.0 %

Table I
PROPORTIONS OF SCENARII WHEN HEURISTICS REACH BEST (OR CLOSE TO BEST) PERFORMANCE, AND AVERAGE DEVIATIONS FROM OPTIMAL MEMORY AND BEST ACHIEVED MAKESPAN.

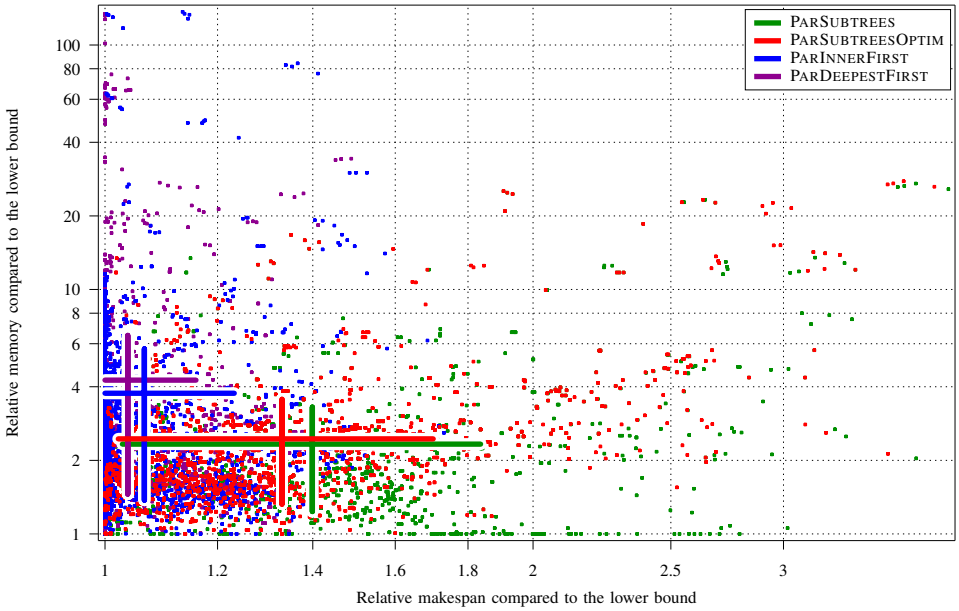

Figure 6. Comparison to lower bounds.

the best heuristics for memory minimization. On average they use less than 2.5 times the amount of memory required by the best sequential postorder (whose memory usage is very close to the optimal sequential memory as noted above), when ParInnerFirst and ParDeepestFirst need respectively 3.7 and 5.2 times this amount of memory. ParInnerFirst and ParDeepestFirst perform best for makespan minimization, having makespans very close on average to the best achieved ones. As the scheduling problem, without memory constraints, is already NP-hard, we do not know what the optimal makespan is. We have seen however that ParInnerFirst and ParDeepestFirst are 2-approximation algorithms for the makespan. Furthermore, given the critical path oriented node ordering, we can expect that ParDeepestFirst's makespan is close to optimal. ParInnerFirst outperforms ParInnerFirst for makespan minimization, at the cost of a noticeable increase in memory. ParSubtrees and ParSubtreesOptim may be better trade-offs, since their average deviation from best makespan is under 35%.

Figures 6, 7, and 8 provide complete results of the simulations. In each figure, a point represent one scenario (one heuristic on one tree with a given number of processors). To better visualize the distribution, we also plot a "cross" for each heuristic: the center of this cross is the average performance, while the branches represent the scope of each objective between the 10th and the 90th percentile of the distribution.

On Figure 6, we plot the results of all simulations compared to some estimations of the lower bounds. The lower bound for memory minimization is the memory usage of the best sequential postorder, which is known to be very close to the optimal sequential traversal. The lower bound for the makespan is the maximum between the total processing time of the tree divided by the number of processors, and the maximum weighted critical path. This figure exhibits the same trends for average values as noted in Table I. When the maximum deviation from the lower bound on the makespan is around 4, the ratio of the parallel memory usage to the optimal sequential one can be far larger, as it is larger than 100 for the extreme cases.

In the following figures, the results of the heuristics is normalized by the results of ParSubtrees (Figure 7) or ParInnerFirst (Figure 8). As expected, ParSub-

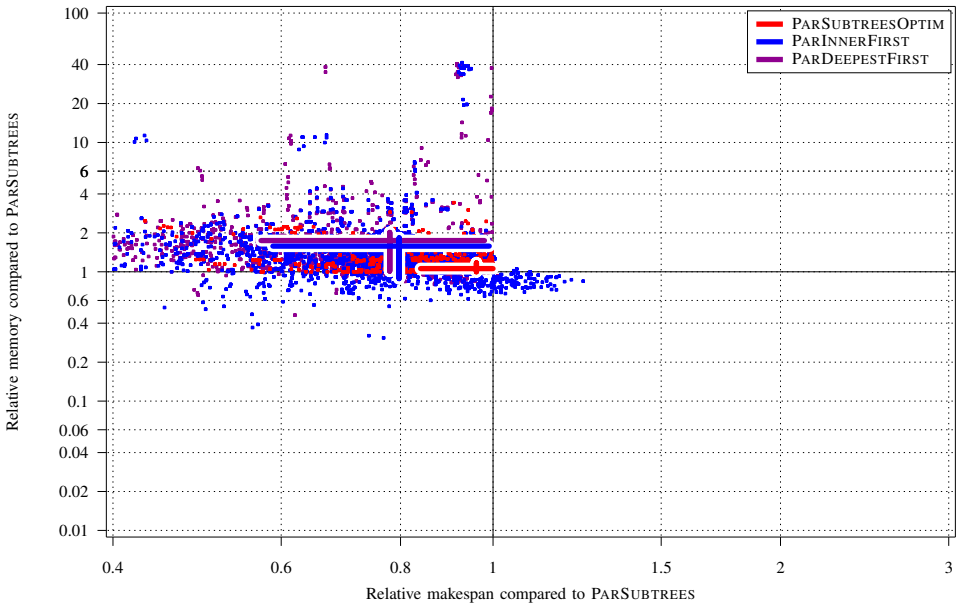

Figure 7. Comparison to ParSubtrees.

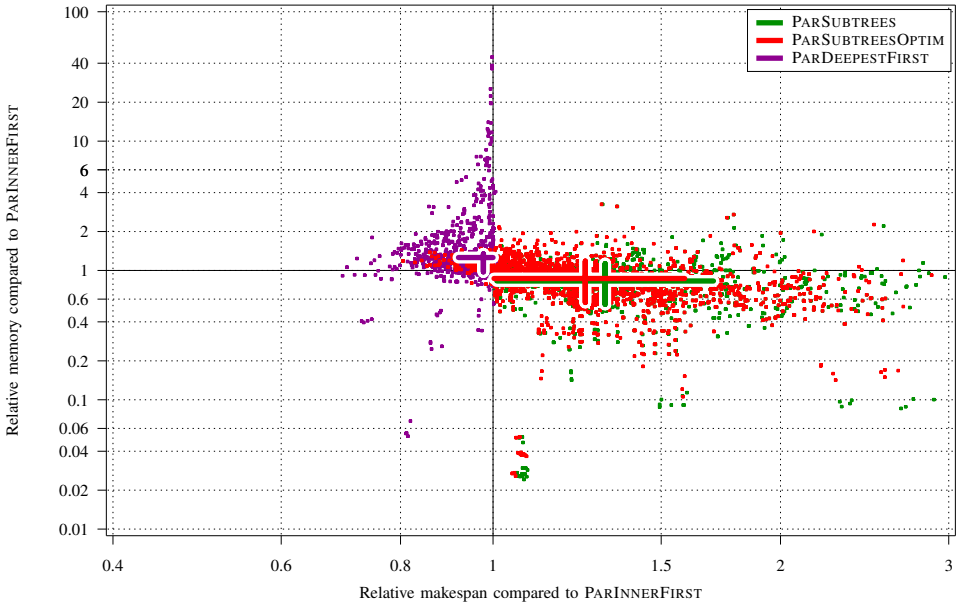

Figure 8. Comparison to ParInnerFirst.

treesOptim gives results close to those of ParSubtrees, with better makespans but slightly worse memory usage. ParDeepestFirst always use more memory than ParInnerFirst, while having comparable makespans. In most cases, ParInnerFirst gives slightly better makespan than ParSubtrees, but uses more memory.

VII. Conclusion

In this study we have shown that the parallel version of the pebble game on trees is NP-complete, hence stressing the negative impact of the memory constraints on the complexity of the problem. More importantly, we have shown that there does not exist any algorithm that is simultaneously an approximation algorithm for both makespan minimization and peak memory usage minimization when scheduling tree-shaped task graphs. We have thus designed heuristics for this problem. We have assess their performance using real task graphs arising from sparse matrices computation. These simulations showed that two of the heuristics, ParSubtrees and ParSubtreesOptim, only needed, for their parallel executions, and on average, 2.5 times the sequential

memory, while achieving makespans that were less than 35% larger than best achieved ones. These heuristics appear thus to deliver interesting trade-offs between memory usage and execution times. In the future work, we will consider designing scheduling algorithms that take as input a cap on the memory usage.

ACKNOWLEDGEMENT

We thank Bora Uçar for his help in creating and managing the data set used in the experiments. We gratefully acknowledge that this work is partially supported by the Marsden Fund Council from Government funding, Grant 9073-3624767, administered by the Royal Society of New Zealand.

REFERENCES

[1] M. Jacquelin, L. Marchal, Y. Robert, and B. Ucar, "On optimal tree traversals for sparse matrix factorization," *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11)*, 2011.

[2] J. W. H. Liu, "On the storage requirement in the out-of-core multifrontal method for sparse factorization," *ACM Trans. Math. Software*, vol. 12, no. 3, pp. 249–264, 1986.

[3] ——, "An application of generalized tree pebbling to sparse matrix factorization," *SIAM J. Algebraic Discrete Methods*, vol. 8, no. 3, 1987.

[4] A. Guermouche and J.-Y. L'Excellent, "Memory-based scheduling for a parallel multifrontal solver," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004, p. 71.

[5] E. Agullo, P. Amestoy, A. Buttari, A. Guermouche, J.-Y. L'Excellent, and F.-H. Rouet, "Robust memory-aware mappings for parallel multifrontal factorizations," 2012, SIAM conf. on Parallel Processing for Scientific Computing (PP12).

[6] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi, "Scheduling data-intensiveworkflows onto storage-constrained distributed resources," in *Proceedings of the IEEE Symposium on Cluster Computing and the Grid (CCGrid'07)*. IEEE, 2007.

[7] C.-C. Lam, T. Rauber, G. Baumgartner, D. Cociorva, and P. Sadayappan, "Memory-optimal evaluation of expression trees involving large objects," *Computer Languages, Systems & Structures*, vol. 37, no. 2, pp. 63–75, 2011.

[8] R. Sethi and J. Ullman, "The generation of optimal code for arithmetic expressions," *J. ACM*, vol. 17, no. 4, pp. 715–728, 1970.

[9] R. Sethi, "Complete register allocation problems," in *Proceedings of the 5th Annual ACM Symposium on Theory of Computing (STOC'73)*. ACM Press, 1973, pp. 182–195.

[10] J. R. Gilbert, T. Lengauer, and R. E. Tarjan, "The pebbling problem is complete in polynomial space," *SIAM J. Comput.*, vol. 9, no. 3, 1980.

[11] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker, "Complexity of machine scheduling problems," *Annals of Discrete Mathematics*, vol. 1, pp. 343–362, 1977.

[12] T. Hu, "Parallel sequencing and assembly line problems," *Operations Research*, vol. 9, 1961.

[13] M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co, 1979.

[14] L. Marchal, O. Sinnen, and F. Vivien, "Scheduling tree-shaped task graphs to minimize memory and makespan," INRIA, Research report 8082, 2012.

[15] J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM Journal of Computing*, vol. 18, no. 2, 1989.

[16] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Technical Journal*, vol. XLV, no. 9, pp. 1563–1581, 1966.

[17] G. Karypis and V. Kumar, *MeTiS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0*, U. of Minnesota, Dpt. of Comp. Sci. and Eng., Army HPC Research Center, Minneapolis, 1998.

[18] J. R. Gilbert, G. L. Miller, and S.-H. Teng, "Geometric mesh partitioning: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 19, no. 6, pp. 2091–2110, 1998.